June, 1996

# Putting Combos and Lists to Work

## These two controls are very powerful, but getting what you want from them isn't always straightforward.

By Tamar E. Granor, Editor

With all the hoopla about the new grid and pageframe controls in Visual FoxPro, there hasn't been much talk about two of the coolest controls around, combo boxes and list boxes. The VFP versions of these old standards are significantly easier to work with and more capable than their FoxPro 2.x counterparts. There are still some complications and a few things that are harder than they should be, though.

This article covers the basics of working with lists and combos and looks at some of the most common issues that arise.

## What are lists and combos?

Both lists and combos are tools for letting a user choose an item from a known list. They differ primarily in appearance, though combos also allow entry of an item not on the list.

Lists are pretty simple. They're known formally as list boxes and are also called scrollable lists. They're common in the Windows environment, appearing in common dialogs like File-Open (for choosing a file) and many other places.

Combos, officially known as combo boxes, come in two flavors: drop-down combo and drop-down list. The key word there is "drop-down" because both types show a single item at a time and open up (or drop down) to show the complete list of choices. Like lists, you'll see combos in lots of Windows dialogs (including File-Open, where a drop-down list shows drives).

The drop-down combo style allows the user to enter a new item while the drop-down list only accepts choices from the listed items.

Although lists and combos are visually different, much of what makes them tick is the same. The discussion below applies equally to lists and combos unless otherwise noted. Throughout this article, the term "list" is used to refer to the actual items displayed in the list or combo.

## List Fundamentals

Lists and combos don't do you much good unless they're populated with something. Visual FoxPro provides no fewer than 10 ways to fill them. About half are pretty useful.

The property that determines the method of filling the list is RowSourceType, which accepts a numeric value from 0 to 9. The value in RowSourceType determines the interpretation of RowSource, which, in most cases, contains the list data or a pointer to it. Table 1 shows the choices for RowSourceType and the interpretation of RowSource in each case.

Table 1 - Combo and List RowSourceTypes. There are 10 different ways to populate the list.

| **RowSourceType** | **RowSource** |
|---|---|
| 0 - None | Left Empty. Items are added individually with code. |
| 1 - Value | Contains a comma-delimited list of items to appear in the list. |
| 2 - Alias | Contains the alias of an open table, cursor or view that provides the data for the list. |
| 3 - SQL Statement | Contains a SQL-SELECT. The cursor created by the query is used to populate the list. |
| 4 - Query | Contains the name of a QPR file. The results of the query fill the list. |
| 5 - Array | Contains the name of an array whose data fills the list. |
| 6 - Fields | Contains the name of one or more fields of a table, cursor or view which populate the list. Only the first field should include the alias. |
| 7 - Files | Contains a file skeleton. The list contains the names of all matching files. |
| 8 - Structure | Contains the alias of a table, cursor or view. The names of its fields fill the list. |
| 9 - Popup | Contains the name of a popup created with DEFINE POPUP. The list contains the bars of the popup. |

The most useful RowSourceTypes are 0 - None, 2 - Alias, 3 - SQL Statement, 5 - Array and 6 - Fields. Alias is most useful if the alias refers to a cursor or view, so fields are in an appropriate order.

When using RowSourceType = 3, be sure to send the query INTO CURSOR something. If the query has no INTO clause, the default BROWSE appears as the form is being instantiated. (The same thing applies to RowSourceType = 4 - Query, too.) Note also that the cursor you create is not automatically closed when the form closes, even if AutoCloseTables is .T. You'll want to close it explicitly in the Destroy method for either the form or the control. Figure 1 shows a form with a combo based on a SELECT. You'll find it on the disk as CounComb.SCX. (All the forms in this article use tables from Visual FoxPro's TasTrade sample database.)

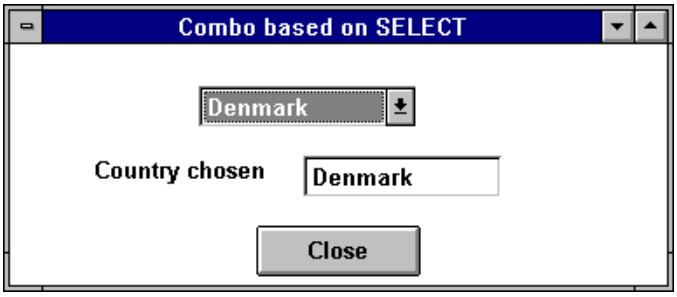


Figure 1. SELECT-based Combo. The combo box of countries has a RowSource which is a SQL-SELECT.

For array-based lists and combos, make the array a property of either the form or the control itself. (You can only make it a property of the control by sub-classing the base list or combo and adding the property in the Class Designer.) Using a property is a better choice than using an array which is a regular memory variable. You'd have to declare the array public in the form to make it available to all the form's methods. (In general, if you find yourself declaring variables public in a form, try making them properties of the form instead.)

Set RowSource to THISFORM.<the array> or THIS.<the array>. For example, the Companion Resource Disk includes a class called arrayCbo, which has a custom property aContents. The RowSource is set to "THIS.aContents".

## Of Indexes and ItemIds

Regardless of the source of the data, lists and combos contain not just one, but two, internal properties for looking at what's actually in the list without knowing anything about the source. Two collections (a collection is like an array), List and ListItem, let you check on the items in the list.

In fact, many properties and methods related to lists and combos come in pairs. The reason is that there are two ways of looking the items in the list. The first, perhaps more intuitive way, is based on their display order. Since lists and combos can be sorted, and, with some RowSourceTypes, items can be added and removed dynamically, the actual display order can vary. The List collection property contains the items in display order. List[1] is the first item shown, List[2] is the second item displayed, and so forth.

But there's another way to look at the items in a list. Each one is assigned its own unique id number (called the "ItemId") when it's added to the list. The id number assigned to a particular entry never changes (though the text of the entry may). The ListItem collection property contains the items in id order.

The difference between the two orders is like the difference between looking at the records in a table in indexed order and in natural order. The List or Index approach corresponds to an indexed table - it shows the items sorted. The ListItem or ItemId approach corresponds to natural (or RECNO()) order. (The analogy falls apart as soon as you allow the table to be packed - there's no such operation for lists.)

The various properties and methods related to lists and combos frequently are paired, having an Index version and an ItemId version. For example, ListIndex tells you which item in Index order is currently highlighted while ListItemId tells you which item in

ItemId order is highlighted. They point to the same item, but when Index order differs from ItemId order, they contain different values. Use ListIndex to get the selected value in the List collection; use ListItemId to do the same for the ListItem collection.

Most of the time, the Index and ItemId versions of a property or method behave the same way (though a few are broken in the initial release of 3.0). However, AddItem and AddListItem are designed to be slightly and subtly different from each other.

The two Add methods let you add items to lists and combos whose RowSourceType is 0-None or 1-Value. The important difference between them is that AddItem *always* adds a new item to the list while AddListItem can either add a new item or change the value of an existing item. This issue is most likely to arise when dealing with multi-column lists, discussed below.

## Storing the Results

Usually, you want to do something with the user's choice from a list or combo. (Not always - sometimes a list may be simply informational.) The list or combo's Value property tells you which item was chosen. ControlSource for the list or combo lets you save the choice directly to a field or variable. However, there's a complication.

As in FoxPro 2.x, the Value of a list or combo can be either character or numeric. If it's character, it behaves as you'd expect - Value contains the text of the chosen item. When Value is numeric, it contains the Index of the chosen item, that is, the position of the item in the list.

In fact, a numeric Value contains the item's Index *even if the data is the list is itself numeric*! Here's what's going on. If you use non-character data as the RowSource for a list, FoxPro internally converts it to character before putting it in the list. (You can check this out by looking at List[1] in the Debug window - note that the value is surrounded by quotes.) By the time the user makes a choice, as far as FoxPro's concerned, you're looking at character data. The same thing is true for other data types, too - put dates in and they turn into characters, too.

By definition, then, a numeric Value indicates you're interested in position, so FoxPro stores the item's Index.

If you assign a ControlSource to a list or combo, it has the same behavior. If it's character, the chosen item's content is stored to the ControlSource; if ControlSource points to a numeric field or variable, the item's Index is saved. (See below for a work-around.)

## Storing a New Value

In drop-down combos, users have the option of typing in a value not on the list. The value entered is stored in the DisplayValue property (and not in Value). In order to save the entry, you need to grab DisplayValue in the combo's Valid event and store it somewhere. You can determine whether a user has typed in a new value by comparing DisplayValue to Value, as follows:

```
IF NOT (THIS.DisplayValue == THIS.Value)
```

```
   * do something with DisplayValue
ENDIF
```

## Saving Non-Character Data

What if you really need to handle numeric or date data in a list or combo and return the actual data? Here's an approach that does the trick. We create a subclass that does what we want (since we want to write and debug this code only once).

The conversion is done in the list or combo's InteractiveChange. Add a custom property called cControlSource which holds, as a character string, the name of the field or variable which is to receive the data from the list. This is really just ControlSource in sheep's clothing.

In the list (or combo) class' InteractiveChange method, put the following code:

```
* Convert character data to appropriate type and store it
* in the field or variable named in cControlSource.
* If cControlSource names a field, it must include
* the alias and a "." to distinguish it from a variable.

LOCAL cValueExpr

DO CASE
CASE TYPE(THIS.cControlSource)$"C,M,U"
   cValueExpr = "THIS.Value"
CASE TYPE(THIS.cControlSource)$"N,B,Y,I"
   cValueExpr = "VAL(THIS.Value)"
CASE TYPE(THIS.cControlSource)="D"
   cValueExpr = "CTOD(THIS.Value)"
CASE TYPE(THIS.cControlSource)="T"
   cValueExpr = "CTOT(THIS.Value)"
ENDCASE

IF "."$THIS.cControlSource
   REPLACE (THIS.cControlSource) WITH &cValueExpr
ELSE
   STORE &cValueExpr TO (THIS.cControlSource)
ENDIF
```

As the comments indicate, there's one restriction on the use of the new class. If cControlSource points to a field, it needs to include the alias of the table. InteractiveChange uses the "." that separates the alias from the field name to determine that it's dealing with a field rather than a variable.

The version above works just fine for lists and for drop-down listboxes. For drop-down combos, there's an added complication - the user might type a new value in rather than choose from the list. As noted above, the data typed in doesn't affect the combo's Value property. Instead, it's stored in DisplayValue. So, we have to force the issue. Add the custom method ForceValue to the combo class with this code:

```
IF NOT (THIS.Value == THIS.DisplayValue)
   THIS.Value = THIS.DisplayValue
   THIS.InteractiveChange()
ENDIF
```

We want to wait for the user to finish entering the new value (rather than acting with each keystroke). There are two ways the user can complete the typed entry - leave the combo or drop it open. To catch the entry as soon as the user is done typing, call ForceEvent from both the combo's Valid and DropDown events, like this:

```
THIS.ForceEvent()
```

The class library ListComb.VCX on this month's Companion Resource Disk contains the classes AnyTypeCombo and AnyTypeList that use this approach to bind any type of value.

## Repopulating a List or Combo

There are times when the data a list or combo is based on changes while the form is running. For example, another user may add a record to the table that list data is based on. At other times, you need to change which data the list is based on. For instance, a list might contain all the suppliers in a particular country and the user chooses the country from a combo. When the user makes a new choice of country, we need to refill the list of suppliers.

In both situations, we use the list or combo's Requery method to update it. Requery tells FoxPro to go back to the original RowSource and repopulate the list with whatever data is now available.

For example, if your RowSource is an array which you fill with a query (as it well might be in the suppliers example above), after you run the query, use the list's Requery method to get the current array contents into the list.

Figure 2 shows a form with a combo of countries and a list of suppliers in that country (based on TasTrade data). Both the combo and the list have RowSourceType set to array. Requery is used for both the combo and the list. (The form is found as Repop.SCX on the disk.)

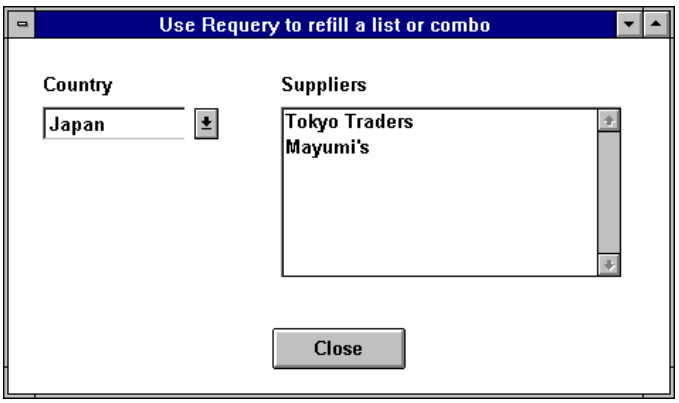


Figure 2. Repopulating a list or combo. The Requery method lets you refill a list or combo when data has changed.

The combo is populated by a query in its own Init that fills the array with a list of the countries. Following the query, we Requery the combo to let it recognize the additions to the array:

```
SELECT DISTINCT Country FROM Supplier ;
   INTO ARRAY THISFORM.aCountries
THIS.Requery()
```

The list uses a similar technique, but takes advantage of a neat trick. The query that fills the list's array needs to be run initially and again whenever the user makes a choice from the combo. So, we put the query itself in the list's Requery method. Then, the list's Init and the combo's InteractiveChange method both call the list's Requery method. Since custom code in a method is executed before the native behavior (in this case, refilling the list), the query runs, then the new data is placed in the list.

Usually, you won't need to put any code in the Requery method. Just calling it is often sufficient to let FoxPro refill the list. Also, don't confuse the Requery method of lists and combos with the REQUERY() function used to re-execute views. They do serve a similar purpose but, in fact, there are times when a call to REQUERY() for a view is followed with a call to a control's Requery method.

## Multiple Columns? No Problem.

In FoxPro 2.x, creating a multi-column list box was a pain. You had to create an array or cursor where all the information for each item was concatenated into a single string and use a vertical bar character (usually CHR(179)) between the individual pieces of information. In Windows or a Mac, you had to use a non-proportional font to get the bars to line up properly. The result was ugly.

In Visual FoxPro, multi-column lists (and combos) are as simple as setting two properties. ColumnCount tells FoxPro how many columns from the RowSource are to appear in the data. ColumnWidths solves the proportional font problem. Best of all, the List and Combo Builders will set both properties (along with a bunch of others) if you let them.

Figure 3 shows a form containing a two-column list based on the Tastrade Products table. The list has RowSourceType set to 6-Fields and RowSource set to "Products.ProductName,UnitPrice". ColumnWidths is set to "210,100" to provide enough room for each field. (This form is on the Companion Disk as MultList.SCX.)

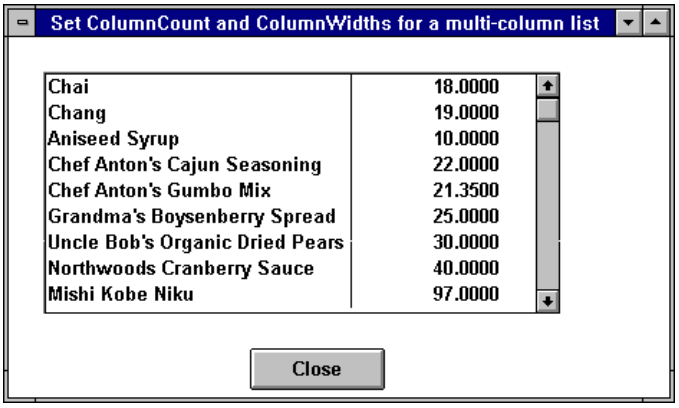


Figure 3. Two-column list. Creating a multi-column list is simple. Just set ColumnCount and ColumnWidths.

Note that ColumnWidths expects a character value containing a comma-delimited list of numbers. In the Property Sheet, you can omit the quotes (as you generally can with character strings there). But when you set ColumnWidths in code, you need to remember that it's a character property.

## Filling a Multi-Column List

It's when populating a multi-column list or combo (with RowSourceType = 0-None) that many people decide that the AddItem method is broken. In fact, this is the situation in which the big difference between AddItem and AddListItem is really apparent.

You can't use AddItem to fill all the columns of a multi-column list. That's because, as noted above, AddItem always adds a whole new item.

There are two ways to populate all the columns. One uses AddListItem. The other takes a more direct approach and addresses the list directly. The key to both methods is the NewItemId property, which always contains the ItemId of the most recently added item.

In either case, AddItem is used for the first column. In the first approach, AddListItem is used for the remaining columns. Figure 4 shows a form (PopMult.SCX) containing a list of TasTrade's Employee table. The following code in the list's Init fills it:

```
SELECT Employee
SCAN
   THIS.AddItem(First_Name)
   THIS.AddListItem(Last_Name,THIS.NewItemId,2)
   THIS.AddListItem(Title,THIS.NewItemId,3)
ENDSCAN
```

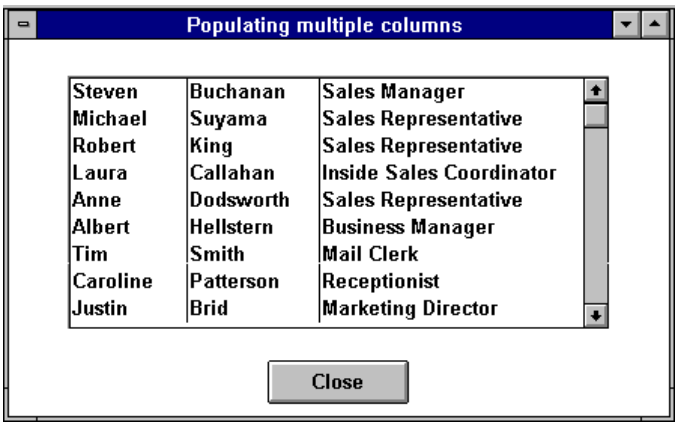


Figure 4 Populating a multi-column list. There are two ways to fill all the columns. AddItem can't do the job all by itself.

The alternative code also uses NewItemId, but addresses the ListItem property directly.

```
SELECT Employee
SCAN
  THIS.AddItem(First_Name)
  THIS.ListItem(THIS.NewItemId,2) = Last_Name
  THIS.ListItem(THIS.NewItemId,3) = Title
ENDSCAN
```

You can use whichever version feels more comfortable to you. (A third approach is to assign the ItemIds yourself with a counter variable. In that case you can use AddListItem for all the columns.)

## Which Column Shall We Save?

Once you have multiple columns in a list, which column's data shows up in the Value and ControlSource? By default, it's the first column. However, you can change it. The BoundColumn property tells Visual FoxPro to put the value from the specified column in Value or ControlSource (assuming they're character, of course). So, it's easy to display five columns and store the value of the third.

Figure 5 shows a form (BindCol.SCX) with a four-column list based on the TasTrade Products table. (The fields shown are ProductId, SupplierId, CategoryId and ProductName.) BoundColumn is set to 4, so that the product name is stored (in this case, to a variable named cProductName specified as the ControlSource for the list).

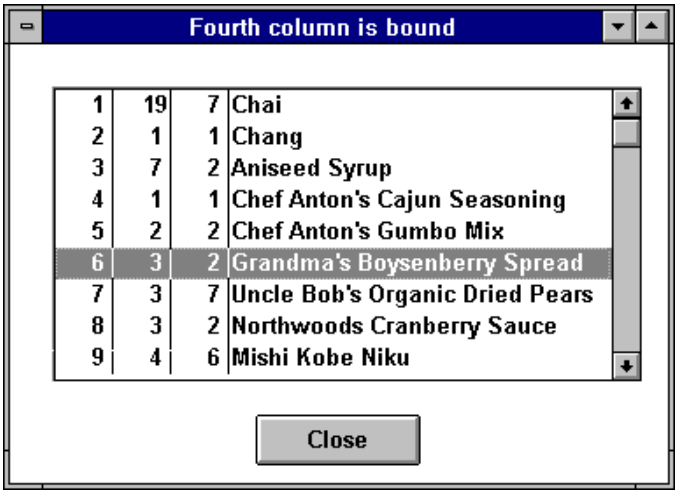


Figure 5. Binding any column. You can specify any column of a multi-column list or combo to be bound to the Value and ControlSource.

This example shows one weakness of the Alias RowSourceType. You rarely want to see columns in the order they appear in the table. There are two solutions to this problem. First, you can use ColumnWidths to make some columns disappear. Just set the width for that column to 0.

A better solution is to use the Fields RowSourceType instead. Then, you can indicate which fields you want and in what order. Normally, you won't show code fields, but only display fields that are meaningful to a user.

When you want to store a code field, but not display it, you can combine the two techniques. Include the code field as a column, but set that column's width to 0. Set BoundColumn to the invisible code column. Figure 6 shows another form (HideCode.SCX) based on the TasTrade Products table. This one has a list showing only the product name and unit price, but the product code is bound to a variable cProductCode. The text box shows the current value of cProductCode, just so you can verify that it's working. The following properties have been set in the Property Sheet for the list (along with the various size and position properties):

```
BoundColumn = 3
ColumnCount = 3
ColumnWidths = 210,110,0
ControlSource = cProductCode
RowSource = products.product_name,unit_price,product_id
RowSourceType = 6
```

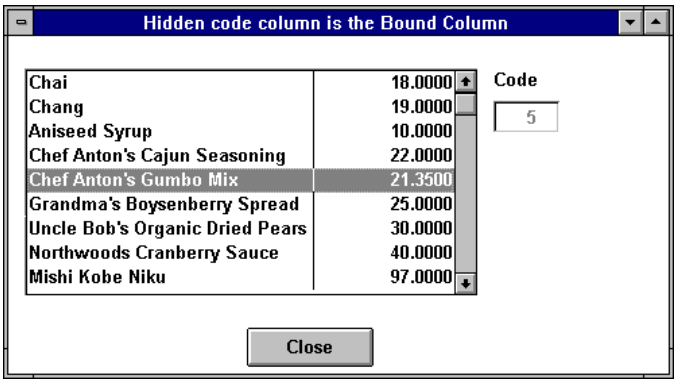


Figure 6. Hiding a column. You can even bind a hidden column (whose width is 0) to the Value or ControlSource.

## What's a DisplayValue?

For combos, the multi-column picture is a little more complex. When a combo is closed, only a single value shows. It's always the first column of the currently selected item. The property DisplayValue contains the text that shows (unless you set DisplayValue to a number, in which case it behaves like Value and holds the Index of the currently selected item).

Because DisplayValue always shows the first column, for combos, the order of the columns in your data matters even more than with lists. You'll rarely use the Alias RowSourceType with combos, unless the alias is for a view. Fields, Query or Array give you more control over what comes first.

## True Combo Boxes

Visual FoxPro's combo boxes are a little different from the standard Windows combos. Throughout Windows, when the user types a value into a combo box, that value is immediately added to the choices in that combo. FoxPro allows the user to type a value without it being added to the list. In fact, that's the default behavior. To get the standard Windows behavior requires a little code.

The exact details of adding the user's input vary depending on the RowSourceType for the combo. The outline is the same in each case, though. In the combo's Valid method, you need code like this:

```
IF NOT (THIS.DisplayValue == THIS.Value)
   * The user has entered a new item.
   * Do something to add the item to the list.
   * For RowSourceType = 0, you'd:
   THIS.AddItem(THIS.DisplayValue)

   * Now make Value point at the item, too.
   * Again, this version works for RowSourceType = 0
```

```
   THIS.Value = ;
      THIS.ListItem[THIS.NewItemId,THIS.BoundColumn]
ENDIF
```

If your combo is based on an array, add a row to the array and put the user's input in it. If you're using one of the table-based RowSourceTypes, you can add a record to the table. Don't forget to Requery the list or combo after you update the RowSource.

You might be inclined to do the updating in the combo's InteractiveChange event, but that one fires for each character the user types. Valid waits until the user finishes typing and moves on. The Companion Resource Disk contains a form (AddCoun.SCX) that uses a simplified version of the code above to let the user add a country to a combo.

## Jumping Ahead with the Keyboard

Both lists and combo's have an IncrementalSearch property that determines what happens when you start typing characters. For combos, IncrementalSearch applies only when you have a drop-down list or when a drop-down combo is open.

When IncrementalSearch is .T., the characters the user types are taken in sequence to narrow the search. So, if a user types "f", then "o", then "x", the highlight will be on the first item beginning "fox". When IncrementalSearch is .F., each character is taken independently. In that case, the same input sequence lands on the first item beginning with "x".

The complicating factor here is that the speed of typing for incremental search is controlled by the system variable _DBLCLICK (which also controls how far apart two clicks can be to constitute a double click). If there's a delay of more than _DBLCLICK between characters, FoxPro assumes you're starting over; otherwise, the new character is added to the current string. The default for _DBLCLICK is .5 for half a second. If you're not seeing incremental search, try typing faster or increasing _DBLCLICK - the maximum is 5.5 seconds.

## Summing up

We've covered a lot of ground here, but believe it or not, this just scrapes the surface. We haven't talked about multi-select or mover lists, or the TopIndex and TopItemId properties, or associating a bitmap with every item in a list. But the information and examples here should get you started experimenting with lists and combos. I think you'll be amazed just how versatile they are.